# Table of Contents

# Problem Statement

Module developers need access to globally accessible resources associated with the EventProcessor invoking their code. In other words, there is a context associated with an EventProcessor instance. That context needs to be accessed from any code attached to that EventProcessor at any time.

The resources to which access is required are singleton-like in that global access is required and are not singleton-like is that there exists one instance reousrce for the EventProcessor in the program. In order to maintain our understanding of each product in the event, these objects will have no effect on the physics performed by the EventProcessor. A few objects of this type have already been identified and are needed now:

- Histogram service
- Timing service
- Message Logging service (footnote: some systems have called this the "ErrorLoggerService", but the need has always been for general than that)

# An Observation

It is likely that most applications that use EventProcessors will only use a single EventProcessor. Nonetheless, we are not permitted to commit to allowing a singleton-like EventProcessor and the ServiceRegistry must accomodate this possibility.

# Definitions

1. Processing module: any instance of a class inhertited from any of edm::Producer, edm::Analyzer, and edm::OutputModule, edm::InputService, edm::ESProducer, and edm::ESSource ("are the last two correct?"). 2. Service class: a facility that performs a well-defined task that is globally acessible and that does not affect physics results. 3. Service object: an instance of the service class. 4. ServiceRegistry: An in-process repository of Service objects who's lifetime is managed by EventProcessor. 5. Service set: a set of service objects associated with a particular EventProcessor.

# Requirements

The configuration of a service object must come from a edm::ParameterSet contained with an EventProcessor edm::ParameterSet.

Code within a processing module must only use the service object type to find it. Services themselves can provide access to named entities they contain.

Service objects must be obtainable from anywhere within processing module code.

The service object obtained by a processing module must be the correct one for its EventProcessor.

The lifetime of a service object is controlled by the lifetime of the service set. The typical lifetime of the service set is controlled by the EventProcessor that it is associated with. Creation of service sets will be limited to the EventProcessor and the main routine.

Service objects are generated before the creation of processing modules within the EventProcessor.

Initialization of a "service set" shall include a list of ParameterSet objects; one object per service object to be included in the service set. Service objects shall be created in the order specified by the list (tranversing the list from begin to end). If a service object in the list requires access to another service further down in the list, the further down service object will be constructed on demand (earlier than specified in the list).

The ServiceRegistry will give thread-safe access to service objects. It is up to the service object to be thread safe.

An exception will be thrown if one of the service objects is accessed outside its proper scope. An example of where such an exception will be thown is setting a static data member service pointer in a processing module.

# Constraints

We will assume that only one EventProcessor will be active per thread at one time. A consequence of this assumption is that if threads are started within an EventProcessor, they will not be able to access the "service set" associated with their EventProcessor. To cope with this, we may allow a unique "service set" to be initialized for the new thread. Adding this extension will depend on if it is necessary. Another way to cope is to allow for attaching subordinate threads to existing EventProcessor "service sets".

Each time a "service set" is activated, the underlying thread may be different.

# Addition Analysis Items

Service objects, upon construction, will be given an opportunity to subscribe to any of a finite set of framework-provided state changes. These state changes fall into three categories, which are explained later in this document. The ServiceRegistry will allow notifications to be disabled or enabled at the level of a category.

The ServiceRegistry will include a method of associating a thread with the currently active service set. This will allow subordinate threads in processing modules to use services from a particular EventProcessor.

Requests have been made to allow the manual injection of service objects into a service set. This is possible provided that the injection happen before any processing modules are created and happen with an EventProcessor or in the main routine.

# Design

All service objects delivered through the service registry must be derived from common base class. A wrapper template will be used to minimize the dependencies in user code. If an external product is to be used as a service, then that product will need to be encapulated in a class that the ServiceRegistry can work with. The encapulating class will do translation from ParameterSet configuration to product-specific configuration.

The SEAL plugin manager will be used to create instances of the service objects.

```
// pseudo-c++

// facade
template <ServiceType>
class Service
{
public:
    Service() { do service registry find to locate object }

    ServiceType* operator->();
    ServiceType& operator*();
};

class ServiceRegistry
{
public:
  static ServiceRegistry& instance();

  template <class ServiceObject> ServiceObject* find();

  class Operate
  {
    Operate(Token)  { setContext   }
    ~Operate(Token) { unsetContext }
  };

  void associatedThreadWithCurrentlyActiveSet(threadid);

private:
  // create a service set described in things and return
  // a handle to it
  Token initializeSet(vector<ParameterSet> things);
  void destroySet(Token);

  friend class Operate;
  friend class edm::EventProcessor;
  friend int main(int argc, char* argv[]);

  void setContext(Token);
  void unsetContext(Token);

  // if necessary to have in this class
  void enableModuleRelatedNotifications();
  void disableModuleRelatedNotifications();
  void enableEventRelatedNotifications();
  void disableEventRelatedNotifications();
};

class Token
{
public:
  explicit Token(int);
  operator int();
private:
```

```
  int key;
};

class ServiceBase
{
public:
  virtual ~ServiceBase();
};

template <class T>
class ServiceWrapper : public ServiceBase
{
public:
  ServiceWrapper(string name, ParameterSet, ActivityRegistry);
  T* getService();
private:
  T service_object_;
};
```

The "initializeSet" call generates a new service set and associates them with the returned token. Each time an EventProcessor becomes active, the "Operate" object will be used to activate a particular set associated with its token (note that sets are active on a per thread basis, so more than one set can be active at the same time). The creation of a token is restricted to the EventProcessor and to the main routine.

Any calls to "find" will be applied to the currently active token for that thread of execution (and associated threads).

These notes to not include all design details. The lifetime management of the token can be controlled by a TokenHolder class, and a shared_ptr to one of these objects can be given out by the "initializeSet" call. Upon destruction of the token holder object, the "destroySet" call could be made.

# Service Constructor Options

The things in this section still need to be discussed. Services, upon creation, can be handed a framework-related object and a parameter set.

## Activity Registry

The "ActivityRegistry" provides a way for services to install callback functions into the event processor paths to get notifications about module activities and state changes.

- Regarding event processing
  - Run begin and end
  - Store begin and end
  - Job begin and end
  - File before/after open and before/after close
- Per processing module
  - entering a module
  - exiting a module
  - useful for services doing things like timing
  - should there is one "functor" instance per module or the same one for all modules?
- State related
  - between two modules
  - beginning of the event loop
  - end of event loop
  - start and end of path?
  - useful for interactive or debugging facilities.
  - should the event/event principal be included as part of the arguments?

# Coding Examples

## Service Registry user

```
class EventProcessor
{
  EventProcessor(...):tok_(ServiceRegistry::instance()->initializeSet(...))
    { }
  EventProcessor(...,Token t):tok_(t)
    { }
  ~EventProcessor()
    { if(own_the_token_)
        serviceRegistry::instance()->destroySet(tok_); }

  // will do "initializeSet" in ctor and "destroySet in dtor
  // tok_ cannot be copied
  ServiceRegistry::TokenHolder tok_;

  void run()
  {
    // activate the set in ctor, deactivate in dtor
    // attached to the current thread
    ServiceRegistry::Operate op(tok_);
    ...
  }
};
```

Note that the example allows for the EventProcess to be constructed using an already esablished token. The lifetime of the token, in this case, should not be controlled by the EventProcessor.

## Service object user

```
class Mod
{
  Mod(): logger_() { }

  void processEvent(Event& e)
  {
    logger_->doStuff();
  }

  Service<Logger> logger_;
};
```

## Service object creator

```
namespace stuff {
 class MyService
 {
 public:
    MyService(const edm::ParameterSet& ps,
              ActivityRegistry& ar)
     {
        ar.wantRunBegin(&MyService::beginRun);
        ar.wantPreFileOpen(&MyService::fileOpen);
        ar.wantBetweenModules(&MyService::between);
        ar.wantEventLoopEnd(&MyService::endLoop);
        ar.wantPreModule(&MyService::preModule);
        ar.wantPostModule(&MyService::postModule);
     }
```

```
    // we do not have the interface for this worked out yet
    // this is a simple example of a crude form that it could take on
    void between(const Worker& before,
                 const Worker& after,
                 const EventPrincipal&);
    void preModule(const ModuleDescription& before);
    void beginRun(const RunNumber&);
 };
}

 using stuff::MyService;
 DEFINE_SEAL_MODULE();
 DEFINE_ANOTHER_FWK_SERVICE(MyService);
```

# Implementation Notes

The boost signal/slots library can probably be used to handle the activity and state change notifications.

-- Main.jbk - 30 Aug 2005

---

This topic: CMS > ServiceRegistryProposal
Topic revision: r4 - 30-Jan-2007 - 08:54:04 - CMSUserSupport